IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In the Matter of the Application of: David M. Chess
Serial No.: 10/696,200 Confirmation No.: 7325
Filed: October 28, 2003
For: System Method and Program Product for Detecting Malicious Software
Examiner: Daniel L. Hoang Group Art Unit: 2136
Attorney Docket No.: GB920030050US1

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF

Sir:

In response to the Office Action of June 4, 2007, having a shortened statutory period for response set to expire on September 4, 2007, and finally rejecting all claims, Applicants respectfully submit this Appeal Brief which is transmitted in triplicate.

Steven E. Bach
Attorney for Applicants
Reg. No. 46,530

Table of Contents

(I) Real Party in Interest

The real party in interest for this Application is assignee INTERNATIONAL BUSINESS MACHINES CORPORATION of Armonk, NY.

(II). Related Appeals and Interferences

There are no related appeals or interferences.

(III). Status of Claims

Claims 1-14 stand rejected under 35 USC 102 as being anticipated by US Patent Number 6,317,868 to Grimm et al. Claims 1-14 are being appealed.

(IV). Status of Amendments

No amendments have been made to the claims.

(V). Summary of claimed subject matter

(V.A) Claims 1 and 14

Claims 1 and 14 are directed to a method (Figs. 2B, 6, 7A-C) and program product (page 15 lines 27-31), respectively for detecting malicious software within or attacking a computer system (page 1 lines 7-9, page 3 lines 13-14, page 4 lines 1-18, page 7 lines 3-10). The method comprises executing a hook routine (step 211 in Fig.2B, 600, 610, 620 Fig. 6, page 4 lines 2-10, page 7 lines 26-31, page 9 lines 19-20, page 11 lines 13-31, page 12 lines 1-30, page 13 lines 7- 30, page 14 lines 7-31, page 15, lines 1-25) in response to a system call (step 206in Fig 2B, steps 601, 611, 621 in Fig. 6, page 4 lines 5-6, page 9 lines 17-20 and 26-28, page 11 lines 13-15 and 27-29 and 31, page 12 lines 1-3 and 5-8 and 10-12 and 14-17, page 14 lines 6-7 and 20-23, page 15 lines 1-5). The hook

routine is located at the location identified by the system call (page 4, lines 5-6, page 7 lines 27-28, page 9 lines 19-20 and 26-28, page 11 lines 13-14 and 27-28 and 31, page 12 lines 1 and 5-6 and 10-11 and 14-16, page 14 lines 6-7 and 20-21, page 15 lines1-2). The hook routine determines a data flow or process requested by the system call (page 4 lines 6-7, page 10 lines 12-24, page 11 lines 13-15 and 27-29, page 12 lines 1-3 and 5-8 and 10-12 and 14-17, page 13 lines 7-13 and 16-21 and 23-26, page 14 lines 7-9 and 21-23, page 15 lines 3-5) and another data flow or process for data related to that of the call (decision steps 603, 613 and 623 in Fig. 6, page 4 line 7, page 10 lines15-19, page 11 lines 30-31, page 12 lines 4-5 and 9-10 and 13-14 and 17-19 and 24-26, page 13 lines 7-13 and 16-21 and 23-26, page 14 lines 9-14 and 23-29, page 15 lines 5-10), and automatically generates a consolidated information flow diagram showing the data flow or process of the call and the other data flow or process (300 in Fig. 3, 330 in Fig. 4, 500 in Fig 5, steps 604, 605, 614, 615, 624, 625 in Fig. 6, page 4 lines 8-9, page 10 lines 12-24, page 11 lines 1-31, page 12 lines 1-28, page 13 lines 7-27, page 14 lines 1 and 14-15 and 27-29, page 15 lines 8-10). Then, the hook routine calls a routine to perform the data flow or process requested by the system call of step 608 (step 212 of Fig. 2B, step 628 of Fig. 6, page 4 lines 9-10, page 11 lines 26-27, page 12 lines 28-30, page 13 lines 27-30, page 14 lines 16-17 and 29-30, page 15 lines 10-11).

(V.B) Claim 8

Claim 8 is directed to an apparatus (Fig. 1B, page 7 lines 19-24) for detecting malicious software within or attacking a computer system. The apparatus comprises a computer system 110 (Fig. 1A, page 7 line 19) with a display screen 120 (Fig. 1, page 7 line 19).

The means for executing a hook routine are the computer system 110 operating a program of instructions (Fig. 2B, Fig. 6, Fig. 7A, Fig. 7B, page 1 lines 7-9, page 3 lines 13-14, page 4 lines 1-18, page 7 lines 3-10). The program comprises executing a hook routine (step 211 in Fig.2B, 600, 610, 620 Fig. 6, page 4 lines 2-10, page 7 lines 26-31, page 9 lines 19-20, page 11 lines 13-31, page 12 lines 1-30, page 13 lines 7- 30, page 14 lines 7-31, page 15, lines 1-25) in response to a system call (step 206 in Fig 2B, steps

601, 611, 621 in Fig. 6, page 4 lines 5-6, page 9 lines 17-20 and 26-28, page 11 lines 13-15 and 27-29 and 31, page 12 lines 1-3 and 5-8 and 10-12 and 14-17, page 14 lines 6-7 and 20-23, page 15 lines 1-5). The hook routine is located at the location identified by the system call (page 4, lines 5-6, page 7 lines 27-28, page 9 lines 19-20 and 26-28, page 11 lines 13-14 and 27-28 and 31, page 12 lines 1 and 5-6 and 10-11 and 14-16, page 14 lines 6-7 and 20-21, page 15 lines1-2). The hook routine determines a data flow or process requested by the system call (page 4 lines 6-7, page 10 lines 12-24, page 11 lines 13-15 and 27-29, page 12 lines 1-3 and 5-8 and 10-12 and 14-17, page 13 lines 7-13 and 16-21 and 23-26, page 14 lines 7-9 and 21-23, page 15 lines 3-5) and another data flow or process for data related to that of the call (decision steps 603, 613 and 623 in Fig. 6, page 4 line 7, page 10 lines15-19, page 11 lines 30-31, page 12 lines 4-5 and 9-10 and 13-14 and 17-19 and 24-26, page 13 lines 7-13 and 16-21 and 23-26, page 14 lines 9-14 and 23-29, page 15 lines 5-10), and automatically generates a consolidated information flow diagram showing the data flow or process of the call and the other data flow or process (300 in Fig. 3, 330 in Fig. 4, 500 in Fig 5, steps 604, 605, 614, 615, 624, 625 in Fig. 6, page 4 lines 8-9, page 10 lines 12-24, page 11 lines 1-31, page 12 lines 1-28, page 13 lines 7-27, page 14 lines 1 and 14-15 and 27-29, page 15 lines 8-10). Then, the hook routine calls a routine to perform the data flow or process requested by the system call of step 608 (step 212 of Fig. 2B, step 628 of Fig. 6, page 4 lines 9-10, page 11 lines 26-27, page 12 lines 28-30, page 13 lines 27-30, page 14 lines 16-17 and 29-30, page 15 lines 10-11).

The means for displaying the information flow diagram is the display screen 120 (Fig. 1, page 7 line 19).

(V.C) Claim 2

Claim 2, which depends from claim 1 is directed to a user monitoring the information flow diagram and comparing the data flow process with a data flow or process expected by the user (page 15 lines 13-25).

(V.D) Claims 3 and 9

Claim 3, which depends from claim 1, and claim 9, which depends from claim 8, are directed to a process and a system, respectively according to their parent claims and wherein the information flow diagram 300, 330, 500 illustrates locations of the data (Fig. 3 - file 301, memory 303, register 304, register 308, second file 309, pages 11, 12; Fig. 4 - file A 310, bytes 311 in memory 312, file B 313, memory 314, register 316, bytes 317 in memory 318, file C 319, memory 320 and socket 321, page 14; Fig. 5 – memory locations 502,504, 506, pages 14, 15) at stages of a processing activity.

(V.E) Claims 4 and 10

Claim 4, which depends from claim 1, and claim 10, which depends from claim 8, are directed to a process and a system, respectively according to their parent claims and wherein the system call (step 205 in Figs. 2A, 2B, page 9 lines 11-12) is selected from the set of: open file, copy file to memory, copy memory to register, mathematical functions, write to file, and network or communication functions (page 8 lines 25-29, page 10 lines 8-10).

(V.F) Claims 5 and 11

Claim 5, which depends from claim 1, and claim 11 which depends from claim 8, are directed to a process and a system, respectively according to their parent claims and wherein the system call is a software interrupt of an operating system (step 202 in Figs. 2A, 2B, page 8 lines 9-18, page 9 lines 8-10 and 17-19, page 10 lines 8-10).

(VI). Grounds of Rejection to be reviewed on appeal

Each of claims 1-14 is rejected under 35 U.S.C. 102 as being anticipated by U.S. Patent No. 6,317,868 to Grimm et al. (hereafter Grimm).

The questions for appeal are whether or not each of claims 1-14 is anticipated by Grimm under 35 U.S.C. 102.

(VI.A) Grouping of Claims for Review

Independent claims 1 and 14 and dependent claims 6, 7 are grouped together for this appeal, as they present the same questions for patentability. These claims rise or fall together.

Independent claims 8 is argued separately as it presents different questions of patentability and does not rise or fall together with claim 1.

Dependent claims 2 is argued separately as it presents different questions of patentability and does not rise or fall together with claim 1.

Dependent claims 3 and 9 are grouped together and argued separately from claims 1 and 8, as they present a different question of patentability and do not rise or fall with claims 1 and 8.

Dependent claims 4 and 10 are grouped together and argued separately from claims 1 and 8, as they present a different question of patentability and do not rise or fall with claims 1 and 8.

Dependent claims 5 and 11 are grouped together and argued separately from claims 1 and 8, as they present a different question of patentability and do not rise or fall with claims 1 and 8.

(VII). Argument

(VII.A) Principles of Law Relating to Anticipation

The Examiner must make a prima facia case of anticipation. "A person shall be entitled to a patent unless. . . (b) the invention was patented or described in a printed publication in this or a foreign country . . . more than one year prior to the date of the application for patent in the United States." 35 U.S.C. 102. It is settled law that each element of a claim must be expressly or inherently described in a single prior art reference to find the claim anticipated by the reference. "A claim is anticipated only if each and every element as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." Verdegaal Bros. v. Union Oil Co. of California, 814 F.3d 63, 631, 2USPQ2d 1051,1053 (Fed. Cir. 1987), cert. denied, 484 U.S. 827 (1987). Inherency, however, may not be established by probabilities or possibilities. The mere fact that a certain thing may result from a given set of circumstances is not sufficient." In re Robertson, 169 F.3d 743, 745, 49 USPQ2d 1949, 1951 (Fed. Cir. 1999)(citations and internal quotation marks omitted). The Examiner has failed to make a prima facia case of anticipation, because the claims on appeal include various elements that are not expressly or implicitly described in the reference cited (i.e., Grimm).

(VII.B) Rejection of Claims 1, 14 under 35 USC 102 over US 6,317,868 (Grim et al.)

Applicants have contended that claims 1 and 14, as originally filed are allowable because they include features that are neither disclosed nor suggested by Grimm or any other references cited in the First Office Action, either individually or in combination.

(VIII.B.1) ***"in response to a system call, executing a hook routine at a location of said system call"***

Claims 1 and 14 include a first element "in response to a system call, executing a hook routine at a location of said system call," and claim 14 includes the similar element "program instructions responsive to a system call for executing a hook routine at a location of said system call."

As described in the present application, in an exemplary embodiment a system call is an operation which transfers control of a processor, such as by stopping the current processing in order to request a service provided by an interrupt handler. The interrupt handler in turn is a program code at a memory location identified by an interrupt vector table. In this example, when a program makes a system call, the system call comprises an interrupt. By use of the interrupt vector table, the system executes the software routine located at the memory location designated (or pointed to) for the particular interrupt in the interrupt vector table. In claim 1, when a program makes a system call during execution of the program, the processor is pointed to a memory location for the system call. The hook routine, as provided in claim 1, is located at the location pointed to by the system call, and is executed in response to the system call. This is significant because the programming level at which software interrupts work is the level at which many computer viruses work (see specification page 8, line 30 to page 9, line 1). Moreover, the present application clearly indicates that it is desirable to hook the lowest level calls where possible (see specification page 9, lines 4-5).

The Examiner has not addressed the substance of this issue, but has instead dismissed the remarks arguing that the remarks in Applicant's response of December 21, 2006 would render the claim ambiguous and indefinite and are not aligned with the specification. The Examiner then reiterated that he interprets intercepting a software component for analysis prior to loading the software in Grim to anticipate the element "in response to a system call, executing a hook routine at a location of said system call" citing column 4, lines 23-27. This passage in Grimm is "When software component 11 as originally created needs to be loaded for execution by a computer, the present invention provides an introspection service 13 that intercepts the software component for analysis."

With respect to the Examiner's argument that the remarks from Applicants response would render the claim ambiguous and indefinite and are not aligned with the specification, it should be noted that the Examiner has improperly taken remarks out of context and tried to treat them as new definitions for claim terms. Specifically, the Examiner cites Applicants remarks at page 5, lines 33-34 "executing a hook routine does not intercept software, but rather transfers control of a processor." This statement is meant to point out the difference between the process of Grimm where software is intercepted and the process of claim 1, in which a hook routine is executed while a program is running in response to a system call in the program. The hook routine itself is defined by steps a-d of claim 1, consistent with the specification. The remark cited by the Examiner is not an attempt to define a hook routine, but to point out that executing a hook routine in response to a system call as claimed as claimed in claim 1 is associated with a transfer of processor control and not associated with intercepting software for analysis.

Grim does not explicitly describe a hook routine, responding to a system call, or that a hook routine is located at the address of a system call. Even if it were possible that introspection service 13 were a hook routine, it is not inherent in Grimm. The introspection service 13 could be any of a number of types of program. Even if it were possible that introspection service 13 were executed in response to a system call, it is not inherent in Grimm. Grimm provides that introspection service 13 intercepts and analyzes software when the software needs to be loaded. Grimm does not provide that the introspection service executes in response to anything, just that it is run before the software is loaded. Even if it is possible that introspection service 13 is located at an address pointed to by a system call, it is not inherent, because introspection service could be located elsewhere. None of these features is required in Grimm. The software could be intercepted in other ways, and therefore the element in question cannot be inherent.

The Examiner has argued that intercepting software before it is loaded is equivalent to hooking system calls. To support this argument, the Examiner points out

that the specification states that later versions of Window, which are not based on DOS operating systems, are based on higher level system calls. Then, Examiner speculates that it won't be possible in these later operating systems to hook the lowest level calls. The portion of the specification referred to in this argument is found in the specification at page 8, line 30 to page 9, line 5 "This low level of programming at which software interrupt operate is the level at which many computer viruses work. Later versions of Windows operating systems include similar functions for hooking operating system calls. Later versions of Windows, which are not based on DOS operating systems, are based on higher level system calls. Low level software interrupts still exist and are available to be hooked. It is desirable in the present invention to hook the lowest level calls where possible."

The important point is that the invention of claims 1 and 14 execute the hooking routine in response to a system call by the operating system. In contrast, Grimm describes intercepting software before it is loaded. Thus, if the introspection service of Grimm executes in response to anything, it is to a command from an operator to load a program not to a system call by the operating system. As clearly shown in an article by Dr. Juergen Haas, a computer scientist and software developer with a Ph.D. in computer Science from the State University of New York at Buffalo, available on "About.com", a system call is "[t]he mechanism used by an application program to request service from the operating system." Similarly, Wikpedia, an online encyclopedia, defines a system call as" the mechanism used by an application program to request service from the operating system." In sharp contrast, Dr. Haas describes load as [a] command [that] loads binary code from a file into the application's address space and calls an initialization procedure in the package to incorporate it into an interpreter." Similarly, in the Hewlett Packard HP3000 Manual, the load command is defined "[t]he LOD command starts the VPLUS forms file definition loading process." Thus, loading software is considered a operation command which is substantially different than a system call, which is a function by which a program requests service from the operating system.

The Examiner has failed to make a prima facia case that the element "in response to a system call, executing a hook routine at a location of said system call," is anticipated by Grimm.

***(VII.B.2)* "determine a dataflow or process requested by said call"**

Claims 1 and 14 include a second element "determine a dataflow or process requested by said call."

In the present application, when a system call is made, the hooking routine executes to monitor and display the operation of the computer system (see specification page 10, lines 1-8). The hooking routines generate an icon or other graphical representation of the current operation to be performed by the original routine (i.e., the routine called by the system call). Thus, in claims 1 and 14, the hooking routine determines the actual function to be performed by the system call (e.g., transferring data, mathematical function, deleting data, copying data, etc.). As pointed out previously, it is important to monitor the system call functions, as this is where many malicious software programs operate. The Examiner cites Grimm at Col. 4, lines 23-27, which describes that an introspection service 13 is provided when software component 11 as originally created needs to be loaded for execution by a computer. This passage does not disclose or suggest that the method of Grimm determines a data flow or process requested by the system call. Grimm does provide that "[i]ntrospection service 13 determines abstractions or object types that are supported by software component as well as operations on these abstractions" (col. 5, lines 7-9). As explained in the present application, data flow or processes are monitored at a system call level. This allows the present invention to track an individual byte of information to monitor and display the operation of the computer without flooding the user with excessive information. Grimm does not provide data flow information to the user, but rather analyzes abstractions of a software component directed at determining security policy measures (e.g., access and authorization) that might apply to the software component. Moreover, the introspection service of Grimm analyzes intercepted software before it is loaded, so it cannot possibly determine the data flow or

process, since the software hasn't even executed at the time of the analysis. At most, introspection service would determine an expected data flow at a software level, not at a system call level.

The Examiner speculates that when a software component is executed by the system, a data process is requested, and that this process is determined because it must be intercepted. In actuality, when a system call is made, a data flow is created. The data flow is the function performed on data (or a file) by the operation of the system call. The data flow that is determined in claims 1 and 14 is the actual function performed by the code being called by the system call. Moreover, software can be intercepted in other ways, without determining the data flow. For example, an operator could manually intercept the software before giving a command to load it.

The Examiner has failed to make a prima facia case of anticipation of the second element, "determine a data flow or process requested by said call."

(VII.B.3) "determine another data flow or process for data related to that of said call"

Claims 1 and 14 include a third element, "determine another data flow or process for data related to that of said call." The present invention provides for associating a current system call function with another system call function by matching file names or memory locations, for example. This step is important to creating a meaningful process flow to track a data flow or process, as operations that in isolation may not appear malicious, when viewed together show a malicious pattern. Grimm does not disclose or suggest stringing together related system call functions to track data flow or processes. The Examiner had suggested in the first office action that this feature is provided in Grimm at column 4, line 65 to column 5, line 2 and at column 6, lines 17-20. However, the cited text addresses the interception and parsing of a software component about to be loaded to identify security policy issues and the security policy operations installed in a modified software component. The Examiner does not appear to have addressed the argument made in the Applicant's response to the first office action.

Grimm does not address associating different system calls that are related by a common file or common data to provide a data flow or process over more than one system call. The Examiner has failed to make a prima facia case of anticipation with respect to the third element "determine a dataflow or process requested by said call."

(VII.B.4) "automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process"

Claims 1 and 14 include a fourth element, "automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process."

The fourth element is directed to the hooking program combining the data flow of a current system call with the data flow of related system calls and presenting this data flow information in a specific useful form, namely a consolidate information flow diagram. A diagram is a graphical representation, as shown in Figs. 3, 4, and 5 and described in the specification at pages 11-15.

Grimm does not disclose or suggest generating a consolidated information flow diagram. In the first office action, the Examiner argued that this element was described by Grimm at column 7, lines 27-31 "A positive response to either of decision blocks 140 or 160 causes an audit record to be created in a block 142 or in a block 162, respectively. In the event that an audit record is necessary, one is created that lists the component operation, its arguments, any access control checks, and their results. First, the audit record is a listing not a diagram. As such, the audit record does not provide an easy means for monitoring data flow during system operation. Second, the audit record is only created on a positive response to specific decision blocks and is therefore not automatically generated. Third, the audit record lists component operation, its arguments, any access control checks, and their results. This is different from a consolidated information flow diagram showing the data flow or process of a system call

and data flow or processes for related system calls. The component operation and arguments are aspects of the software being analyzed, not the data flow created by system calls by the operating system. Access control checks and their results are related to determining security policy measures (e.g., access and authorization).

The Examiner has failed to make a prima facia case of anticipation with respect to the fourth element, "automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process."

(VII.C) Rejection of Claim 8 under 35 USC 102 over 6,317,868 (Grim et al.)

Applicants have contended that claim 8, as originally filed is allowable because it includes features that are neither disclosed nor suggested by Grimm or any other references cited in the First Office Action, either individually or in combination.

(VII.C.1) ***elements previously discussed under claim 1***

"means responsive to a system call, for executing a hook routine at a location of said system call"

This element is directed to a computer operating a program of instruction substantially similar to the first element argued under claims 1 and 14. These arguments will not be repeated here.

"determine a dataflow or process requested by said call"

This element is directed to a computer operating a program of instruction substantially similar to the second element argued under claims 1 and 14. These arguments will not be repeated here.

"determine another data flow or process for data related to that of said call"

This element is directed to a computer operating a program of instruction substantially similar to the third element argued under claims 1 and 14. These arguments will not be repeated here.

"automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process"

This element is directed to a computer operating a program of instruction substantially similar to the fourth element argued under claims 1 and 14. These arguments will not be repeated here.

(VII.C.2) *"means for displaying said information flow diagram"*

Claim 8 includes the element "means for displaying said information flow diagram." The structure associated with this element is the display screen 120. Thus, the information flow diagram representing the information flow determined by computer 110 is displayed on display screen 120. Grimm does not describe either an information flow diagram or a display screen. Accordingly, the Examiner has failed to make a prima facia case of anticipation.

(VII.D) Rejection of Claim 2 under 35 USC 102 over 6,317,868 (Grim et al.)

Applicants have contended that claim 2, as originally filed is allowable independently of its parent claim 1, because it includes features that are neither disclosed nor suggested by Grimm or any other references cited in the First Office Action, either individually or in combination.

(VII.D.1) "a user monitors said information flow diagram"

Claim 2 includes the element "a user monitors said information flow diagram." Grimm does not disclose or suggest a user monitoring an information flow diagram. A user monitoring the information flow diagram is significant because it allows the user to detect suspicious activity in real time and take remedial measures (page 15 lines 13-25).

The Examiner determined in error that this element is provided by Grimm at column 6, lines 58-63. Grimm provides that the enforcement service performs access checks on each argument or object to be passed to the component operation. Grimm neither discloses nor suggests an information flow diagram. Nor does Grimm provide that a user monitors the information flow diagram. In fact, Grimm does not disclose or suggest any monitoring by a user. Moreover the access check of Grimm is performed before the software is loaded and therefore does not suggest real-time monitoring of a data flow diagram. Accordingly, the Examiner has failed to make a prima facia case of anticipation.

(VII.D.2) *"compares the data flow process of steps (a) and (b) with a data flow or process expected by said user"*

Claim 2 includes the feature "compares the data flow or process of steps (a) and (b) with a data flow or process expected by said user." Grimm is directed to performing access checks for a subject and component code, protection domain transfers, and auditing based on the subject and the modified software. Grimm does not disclose or suggest comparing data flow during execution of a system call with the expected dataflow for the system call. Accordingly, the Examiner has failed to make a prima facia case of anticipation.

(VII.E) Rejection of Claims 3, under 35 USC 102 over 6,317,868 (Grim et al.)

Applicants have contended that claims 3 and 9, as originally filed are allowable independently of their parent claims 1 and 8, respectively, because they include a feature

that is neither disclosed nor suggested by Grimm or any other references cited in the First Office Action, either individually or in combination.

(VII.E.1) *"said information flow diagram illustrates locations of said data at stages of a processing activity"*

Claims 3 and 9 include the element "said information flow diagram illustrates locations of said data at stages of a processing activity." The present invention provides an embodiment in which the flow of data is captured and the location of the data at stages of processing is illustrated. This allows a user to quickly see where data is being transferred to, and therefore, whether data is being manipulated in an undesirable way. Grimm does not disclose or suggest illustrating locations of data during processing.

The Examiner has concluded in error that the audit record of Grimm is an information flow diagram that illustrates locations of data at stages of a processing activity. Contrary to this conclusion, Grimm defines an audit record "[an audit record] is created that lists the component operation, its arguments, any access control checks, and their results". Grimm defines 'audit record' as a listing, which is fundamentally different from a flow diagram. A flow diagram as explained in Applicants' specification is a graphic representation of data flow which will reveal suspicious data flow in a timely manner to allow remedial action (see page 15 lines 13-25).

Moreover the definition of an 'audit record' in Grimm is not consistent with the Examiner's conclusion. The component operation refers to a software operation or command. The arguments are fields or objects of the operation. Access control checks are security policy provisions. None of the information identified in the definition of 'audit record' in Grimm would provide the actual location of a particular data item at stages of a processing activity. Accordingly, the Examiner has failed to make a prima facia case of anticipation.

(VII.F) Rejection of Claims 4, 10 under 35 USC 102 over 6,317,868 (Grim et al.)

Applicants have contended that claims 4 and 10, as originally filed are allowable independently of their parent claims 1 and 8, respectively, because they include a feature that is neither disclosed nor suggested by Grimm or any other references, either individually or in combination.

(VII.F.1) *"said system call is selected from the set of: open file, copy file to memory, copy memory to register, mathematical functions, write to file, and network or communication functions"*

Claims 4 and 10 include the element "said system call is selected from the set of: open file, copy file to memory, copy memory to register, mathematical functions, write to file, and network or communication functions." Grimm does not disclose or suggest a hook routine executing in response to any of the claimed system calls. In the first office action the Examiner appeared to suggest in error that this feature is disclosed by the statement in Grimm "Based upon information determined by introspection service 13, a security policy service 15 instructs an interposition service 17, which is also included in the present invention, how to modify the original software component to adhere to the security policies of the site."

Applicants respectfully contended that modification of incoming software by adding access checks, protection domain transfers, and auditing does not disclose or suggest that a hook routine executes in response to one of the claimed system calls.

In the final office action, the Examiner suggests in error that this feature is described in the Background section of Grimm at column 1 lines 58-67 and column 2 lines 1-16.

> It would clearly be desirable to provide security administrators with a mechanism allowing them to control and observe the behavior of software components derived from a different source in regard to security and access issues. The control and observation of these components should

> thus be independent of the origin of the component, and independent of the security services and hosting of the hosting component system or operating system used to execute the software component. Further, it would be desirable for this mechanism to interpose access control checks, protection domain transfers, and auditing onto software component operations in a transparent manner that does not otherwise affect the functionality and execution of the software components. The auditing may encompass instrumenting the software components to provide information relating to the execution of the component as thus modified. Such information might provide an indication of the efficiency of the software component in completing a function, or the processor overhead that the software component creates, or indicate the number of times that it calls a routine, etc. The mechanism providing these functions should also separate the enforcement and auditing of the security policy from the actual site-specific security policy. By doing so, the approach should be appropriate for use in environments that rely on potentially insecure software components in which security policies frequently change.

Claims 4 and 10 provide a specific group of system calls at which to locate hooking programs to effectively monitor data flow and processes. The cited passage from Grimm is a description of what is viewed as lacking in the prior art. This passage does not describe inserting hooking programs at the location of system calls, much less the specific group of system calls in claims 4 and 10. Accordingly, the Examiner has failed to make a prima facia case of anticipation.

(VII.G) Rejection of Claims 5, 11 under 35 USC 102 over 6,317,868 (Grim et al.)

Applicants have contended that claims 5 and 11, as originally filed are allowable independently of their parent claims 1 and 8, respectively, because they include a feature that is neither disclosed nor suggested by Grimm or any other references, either individually or in combination.

(VII.G.1) *"said system call is a software interrupt of an operating system."*

Claims 5 and 11 include the element "said system call is a software interrupt of an operating system." As with claim 4, the office action seems to suggest that modification of incoming software by adding access checks, protection domain transfers, and auditing

either discloses or suggests a system call that is a software interrupt of an operating system. However, Grimm does not disclose or suggest executing a hooking routine in response to a system call that is a system interrupt of the operating system. Accordingly, the Examiner has failed to make a prima facia case of anticipation.

(VIII) Claims Appendix

Listing of Claims:

1. (original) A method for detecting malicious software within or attacking a computer system, said method comprising the steps of:

In response to a system call, executing a hook routine at a location of said system call to (a) determine a data flow or process requested by said call, (b) determine another data flow or process for data related to that of said call, (c) automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process, and after steps (a-c), (d) call a routine to perform said data flow or process requested by said call.

2. (original) A method as set forth in claim 1, wherein a user monitors said information flow diagram and compares the data flow process of steps (a) and (b) with a data flow or process expected by said user.

3. (original) A method as set forth in claim 1, wherein said information flow diagram illustrates locations of said data at stages of a processing activity.

4. (original) A method as set forth in claim 1, wherein said system call is selected from the set of: open file, copy file to memory, copy memory to register, mathematical functions, write to file, and network or communication functions.

5. (original) A method as set forth in claim 1, wherein said system call is a software interrupt of an operating system.

6. (original) A method as set forth in claim 1, wherein said system call causes a processor to stop its current activity and execute said hook routine.

7. (original) A method as set forth in claim 1 wherein said system call is made by malicious software.

8. (original) A system for detecting malicious software in a computer system, said system comprising:

means, responsive to a system call, for executing a hook routine at a location of said system call to (a) determine a data flow or process requested by said call, (b) determine another data flow or process for data related to that of said call, (c) automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process, and after steps (a-c), (d) call a routine to perform said data flow or process requested by said call; and

means for displaying said information flow diagram.

9. (original) A system as set forth in claim 8, wherein said information flow diagram illustrates locations of said data at stages of a processing activity.

10. (original) A system as set forth in claim 8, wherein said system call is selected from the set of: open file, copy file to memory, copy memory to register, mathematical functions, write to file, and network or communication functions.

11. (original) A system as set forth in claim 8, wherein said system call is a software interrupt of an operating system.

12. (original) A system as set forth in claim 8, wherein said system call causes a processor to stop its current activity and execute said hook routine.

13. (original) A system as set forth in claim 8 wherein said system call is made by malicious software.

14. (original) A computer program product for detecting malicious software in a computer system, said computer program product comprising:

a computer readable medium;

program instructions, responsive to a system call, for executing a hook routine at a location of said system call to (a) determine a data flow or process requested by said call, (b) determine another data flow or process for data related to that of said call, (c) automatically generate a consolidated information flow diagram showing said data flow or process of said call and said other data flow or process, and after steps (a-c), (d) call a routine to perform said data flow or process requested by said call; and wherein

said program instructions are recorded on said medium.

(IX). Evidence appendix

IX.A.

Haas, Dr. Juergen, "system Call", About.com, printed July 30, 2007 (http://about.com/cs/linux101/g/system_call.htm?terms=system+call)

IX.B.

"System Call", Wikpedia, printed July 30, 2007 (http://Wikipedia.org/Wiki/System_call)

IX.C.

Haas, Dr. Juergen, "Linux/Unix Command: *load*", About.com, printed July 30, 2007 (http://linux.about.com/library/cmd/blcmdln_load.htm)

IX.D.

"LOAD command", HP SYSTEM DICTIONARY XL UTILITIES-PART 3 SDVPD, printed July 30, 2007 (http://docs.hp.com/cgi-bin/doc3k/B322569003.10481/17)

About.com : Focus on Linux

"system call"

From Juergen Haas,
Your Guide to Focus on Linux.

Definition: system call: The mechanism used by an application program to request service from the operating system. System calls often use a special machine code instruction which causes the processor to change mode (e.g. to "supervisor mode" or "protected mode"). From Linux Guide @FirstLinux

* **Linux/Unix/Computing Glossary**

System call

From Wikipedia, the free encyclopedia

In computing, a **system call** is the mechanism used by an application program to request service from the operating system.

Contents

Background

In addition to processing data in its own memory space, an application program might want to use data and services provided by the system. Examples of the system providing a service to an application include: reporting the current time, allocating memory space, reading from or writing to a file, printing text on screen, and a host of other necessary actions.

Since the machine and its devices are shared between all the programs, the access must be synchronized. Some of the activities may fail the system or even destroy something physically. For these reasons, the access to the physical environment is strictly managed by the BIOS and operating system. The code and data of the OS is located in a protected area of memory and cannot be accessed/damaged by user applications. The only gate to the hardware is system calls, which are defined in the operating system. These calls check the requests and deliver them to the OS drivers, which control the hardware input/output directly.

Modern processors can typically execute instructions in several, very different, privileged states. In systems with two levels, they are usually called user mode and supervisor mode. Different privilege levels are provided so that operating systems can restrict the operations that programs running under them can perform, for reasons of security and stability. Such operations include accessing hardware devices, enabling and disabling interrupts, changing privileged processor state, and accessing memory management units.

With the development of separate operating modes with varying levels of privilege, a mechanism was needed for transferring control safely from lesser privileged modes to higher privileged modes. Less privileged code could not simply transfer control to more privileged code at any arbitrary point and with any arbitrary processor state. To allow it to do so could allow it to break security. For instance, the less privileged code could cause the higher privileged code to execute in the wrong order, or provide it with a bad stack.

Mechanism

System calls often use a special CPU instruction which causes the processor to transfer control to more privileged code, as previously specified by the more privileged code. This allows the more privileged code to specify where it will be entered as well as important processor state at the time of entry.

When the system call is invoked, the program which invoked it is interrupted, and information needed to continue its execution later is saved. The processor then begins executing the higher privileged code, which, by examining processor state set by the less privileged code and/or its stack, determines what is being requested. When it is finished, it returns to the program, restoring the saved state, and the program continues executing.

Note that in many cases, the actual return to the program may not be immediate. If the system call performs any kind of lengthy I/O operation, for instance disk or network access, the program may be suspended ("blocked") and taken off the "ready" queue until the operation is complete, at which point the operating system will again make it a candidate for execution.

The library as an intermediary

Generally, operating systems provide a library that sits between normal programs and the rest of the operating system, usually the C library (libc), such as glibc and the Microsoft C runtime. This library handles the low-level details of passing information to the kernel and switching to supervisor mode, as well as any data processing and preparation which does not need to be done in privileged mode. Ideally, this reduces the coupling between the operating system and the application, and increases portability.

On exokernel based systems, the library is especially important as an intermediary. On exokernels, OSes shield user applications from the very low level kernel API, and provide abstractions and resource management.

Examples and tools

On POSIX and similar systems, popular system calls are `open`, `read`, `write`, `close`, `wait`, `exec`, `fork`, `exit`, and `kill`. Many of today's operating systems have hundreds of system calls. For example, Linux has 319 different system calls. FreeBSD has about the same (almost 330).

Tools such as strace and truss report the system calls made by a running process.

In the Java platform, there is no need for the Java virtual machine to interrupt itself to make the system call safer. The effect is reached by providing a higher level of isolation by renunciation of arbitrary memory pointers, which allows the safe placement of all the code into one memory space. The Java virtual machine is indistinguishable from its supporting library Java Runtime Environment in this platform where API consists of a set of system objects, invoking methods of which system calls are made, and no privileged instructions are needed. A similar approach is used by Microsoft in its .Net platform and Singularity OS.

Typical implementations

Implementing system calls requires a control transfer which involves some sort of architecture specific

feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the kernel so software simply needs to set up some register with the system call number they want and execute the software interrupt. Linux uses this implementation on x86 where the system call number is placed in the EAX register before interrupt 0x80 is executed.

For many RISC processors this is the only feasible implementation, but CISC architectures such as x86 support additional techniques. One example is SYSCALL/SYSRET which is very similar to SYSENTER/SYSEXIT (the two mechanisms were created by Intel and AMD independently, but do basically the same thing). These are "fast" control transfer instructions that are designed to quickly transfer control to the kernel for a system call without the overhead of an interrupt.

An older x86 mechanism is called a call gate and is a way for a program to literally call a kernel function directly using a safe control transfer mechanism the kernel sets up in advance. This approach has been unpopular, presumably due to the requirement of a far call which uses x86 segmentation and the resulting lack of portability it causes, and existence of the faster instructions mentioned above.

External links

- Linux system calls (http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html) - system calls for Linux kernel 2.2, with IA32 calling conventions
- How System Calls Work on Linux/i86 (http://www.tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html)
- Sysenter Based System Call Mechanism in Linux 2.6 (http://manugarg.blogspot.com/2006/07/sysenter-based-system-call-mechanism.html)

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

Retrieved from "http://en.wikipedia.org/wiki/System_call"

Categories: Operating system technology | Application programming interfaces

Linux / Unix Command: *load*

Command Library

NAME

load - Load machine code and initialize new commands.

SYNOPSIS

load *fileName*
load *fileName packageName*
load *fileName packageName interp*

DESCRIPTION

This command loads binary code from a file into the application's address space and calls an initialization procedure in the package to incorporate it into an interpreter. *fileName* is the name of the file containing the code; its exact form varies from system to system but on most systems it is a shared library, such as a **.so** file under Solaris or a DLL under Windows. *packageName* is the name of the package, and is used to compute the name of an initialization procedure. *interp* is the path name of the interpreter into which to load the package (see the **interp** manual entry for details); if *interp* is omitted, it defaults to the interpreter in which the **load** command was invoked.

Once the file has been loaded into the application's address space, one of two initialization procedures will be invoked in the new code. Typically the initialization procedure will add new commands to a Tcl interpreter. The name of the initialization procedure is determined by *packageName* and whether or not the target interpreter is a safe one. For normal interpreters the name of the initialization procedure will have the form *pkg*_**Init**, where *pkg* is the same as *packageName* except that the first letter is converted to upper case and all other letters are converted to lower case. For example, if *packageName* is **foo** or **FOo**, the initialization procedure's name will be **Foo_Init**.

If the target interpreter is a safe interpreter, then the name of the initialization procedure will be *pkg*_**SafeInit** instead of *pkg*_**Init**. The *pkg*_**SafeInit** function should be written carefully, so that it initializes the safe interpreter only with partial functionality provided by the package that is safe for use by untrusted code. For more information on Safe-Tcl, see the **safe** manual entry.

The initialization procedure must match the following prototype:

```
typedef int Tcl_PackageInitProc(Tcl_Interp *interp);
```

The *interp* argument identifies the interpreter in which the package is to be loaded. The

initialization procedure must return **TCL_OK** or **TCL_ERROR** to indicate whether or not it completed successfully; in the event of an error it should set the interpreter's result to point to an error message. The result of the **load** command will be the result returned by the initialization procedure.

The actual loading of a file will only be done once for each *fileName* in an application. If a given *fileName* is loaded into multiple interpreters, then the first **load** will load the code and call the initialization procedure; subsequent **load**s will call the initialization procedure without loading the code again. It is not possible to unload or reload a package.

The **load** command also supports packages that are statically linked with the application, if those packages have been registered by calling the **Tcl_StaticPackage** procedure. If *fileName* is an empty string, then *packageName* must be specified.

If *packageName* is omitted or specified as an empty string, Tcl tries to guess the name of the package. This may be done differently on different platforms. The default guess, which is used on most UNIX platforms, is to take the last element of *fileName*, strip off the first three characters if they are **lib**, and use any following alphabetic and underline characters as the module name. For example, the command **load libxyz4.2.so** uses the module name **xyz** and the command **load bin/last.so {}** uses the module name **last**.

If *fileName* is an empty string, then *packageName* must be specified. The **load** command first searches for a statically loaded package (one that has been registered by calling the **Tcl_StaticPackage** procedure) by that name; if one is found, it is used. Otherwise, the **load** command searches for a dynamically loaded package by that name, and uses it if it is found. If several different files have been **load**ed with different versions of the package, Tcl picks the file that was loaded first.

PORTABILITY ISSUES

Windows

When a load fails with "library not found" error, it is also possible that a dependent library was not found. To see the dependent libraries, type ``dumpbin -imports <dllname>" in a DOS console to see what the library must import. When loading a DLL in the current directory, Windows will ignore ``./" as a path specifier and use a search heuristic to find the DLL instead. To avoid this, load the DLL with

```
load [file join [pwd] mylib.DLL]
```

SEE ALSO

info sharedlibextension, Tcl_StaticPackage(3), safe(n)

KEYWORDS

binary code, loading, safe interpreter, shared library

Important: Use the *man* command (*% man*) to see how a command is used on your particular computer.

>> Linux/Unix Command Library

>> Shell Command Library

HP SYSTEM DICTIONARY XL UTILITIES-Part 3 SDVPD

LOAD Command

The LOAD command starts the VPLUS forms file definition loading process. To load specific forms from the forms file, specify the form names after the command. To load all forms definitions from the forms file, issue the command with no forms names following it.

Syntax

LOAD [formname1formname2... formname3] {.}

Parameters:

formname:Name of the VPLUS forms to be loaded

(X). Related proceedings appendix

There are no related proceedings.